```
-- CODING CONVENTIONS:
-- No globals. Line length: 80 chars or less.
-- Parse settings from a help string (see top of file).
-- This code does no run anything. Rather it is a module to be loaded
-- and run by e.g. rlgo.lua)
--
-- VARIABLE NAME CONVENTIONS:
-- Leading__upper_case : class
-- i.                  : instance var
-- l. s                : reference to a library function
-- prefix _            : some internal function,variable.
--
-- TYPE HINT CONVENTIONS (where practical, on function arguments):
-- - t = table
-- - prefix s=string
-- - prefix n=num
-- - prefix is=boolean
-- - class names in lower case denote vars of that class
-- - suffix s denotes table of things
local l = require"lib"
local the = l.settings[[

RL.LUA : stings
(c)2022 Tim Menzies <timm@ieee.org> BSD(2clause).

USAGE:
  lua rlgo.lua [ -bFghksS [ARG] ]

OPTIONS:
 -b  --bins    discretization control        = 8
 -F  --Far     in "far", how far to seek     = .95
 -g  --go      start-up action               = pass
 -h  --help    show help                     = false
 -k  --keep    keep only these nums          = 512
 -m  --Min     stop at N^m                   = .5
 -p  --p       distance coefficient          = 2
 -s  --seed    random number see             = 10019
 -S  --Some    in "far", how many to search = 512
]]
local RL   = {About={}, Data={}, Row={},Col={},the=the}
local About= RL.About -- factory for making columns
local Data = RL.Data -- store rows, and their column summaries
local Row  = RL.Row -- stores one row.
local Col  = RL.Col -- summarize 1 column. Has 2 roles-- NOMinal,RATIO for syms,nums

-- FYI: I considered splitting Col into two (one for NOMinals and one for
-- RATIOs).  But as shown in Col (below), one of those two cases can usually be
-- handled as a one-liner. So the benefits of that reorg is not large.
--
-- One nuance here is that, to save memory, Rows are created by the FIRST Data
-- that sees a record, then shared across every other clone  of the data (e.g.
-- when clustering, the super Data points to the same Row as the sub-Data
-- cluster of all the other rows closest to that first Row).  Since  rows
-- maintains a pointer to its creator Data object, that first data Data can be
-- used to store information about the entire data spaces (e.g. the max and min
-- possible values for each columns).  This makes certain functions easier like,
-- say, distance).
--
--      ╔═╗╔═╗╔═╗╔╦╗
--      ╠═╣╠╣ ║ ║ ║
--      ╩ ╩╚  ╚═╝ ╩
--
-- Factory for making columns.
function About.new(sNames)
  return About._cols({names=sNames, all={}, x={}, y={}, klass=nil},sNames) end

-- How to recognize different column types
local _is={
  nom   = "^[a-z]",  -- ratio cols start with uppercase
  goal  = "[!+-]$",  -- !=klass, [+,-]=maximize,minimize
  klass = "!$",      -- klass if "!"
  skip  = ":$",      -- skip if ":"
  less  = "-$"}      -- minimize if "-"

-- Turn a list of column names into Col objects. If the new col is independent
-- or dependent or a goal attribute then remember that in i.x or i.y or i.klass.
function About._cols(i,sNames)
  for at,name in pairs(sNames) do
    local col = l.push(i.all, Col.new(name,at))
    if not name:find(_is.skip) then
      l.push(name:find(_is.goal) and i.y or i.x, col)
      if name:find(_is.klass) then i.klass=col end end end
  return i end

-- Update, only the non-skipped cols (i.e. those found in i.x and j.x.
function About.add(i,t)
  local row = t.cells and t or Row.new(i, t)
  for _,cols in pairs{i.x,i.y} do
    for _,coll in pairs(cols) do
      Col.add(coll, row.cells[coll.at]) end end
  return row end
--
--      ╦═╗╔═╗╦ ╦
--      ╠╦╝║ ║║╔╝
--      ╩╚═╚═╝╚╝
--
-- Hold one record
function Row.new(about,t)
  return {_about=about, cells=t, cooked=l.map(t,l.same)} end

-- Everything in rows, sorted by distance to i.
function Row.around(i,rows)
  local fun = function(j) return {row=j, d=Row.dist(i,j)} end
  return l.sort(l.map(rows, fun), l.lt"d") end

-- Recommend sorting i before j (since i is better).
function Row.better(i,j)
  i.evaled,j.evaled= true,true
  local s1,s2,d,n,x,y=0,0,0,0
  local ys,e = i._about.y,math.exp(1)
  for _,col in pairs(ys) do
    x,y= i.cells[col.at], j.cells[col.at]
    x,y= Col.norm(col,x), Col.norm(col,y)
    s1 = s1 - e^(col.w * (x-y)/#ys)
    s2 = s2 - e^(col.w * (y-x)/#ys)
  end
  return s1/#ys < s2/#ys end

-- Distance
function Row.dist(i,j)
  local d,n,x,y,dist1=0,0
  local cols = cols or i._about.x
  for _,col in pairs(cols) do
    x,y = i.cells[col.at], j.cells[col.at]
    d   = d + Col.dist(col,x,y)^the.p
    n   = n + 1 end
  return (d/n)^(1/the.p) end

--
--      ╔═╗╔═╗╦
--      ║  ║ ║║
--      ╚═╝╚═╝╩═╝
--
-- Summarize one column.
function Col.new(txt,at)
  txt = txt or ""
  return {n    = 0,                  -- how many items seen?
          at   = at or 0,            -- position ot column
          txt  = txt,                -- column header
          isNom= txt:find(_is.nom),
          w    = txt:find(_is.less) and -1 or 1,
          ok   = true,               -- false if some update needed
          _has  = {}} end            -- place to keep (some) column values.

-- Create columns with particular roles.
function Col.ratio(...) local i=Col.new(...); i.isNom=false; return i end
function Col.nom(...)   local i=Col.new(...); i.isNom=true;  return i end

-- Update. Optically, repeat n times.
function Col.add(i,x,  n)
  if x ~= "?" then
    n = n or 1
    i.n = i.n + n
    if i.isNom then i._has[x] = n + (i._has[x] or 0) else
      for _ = 1,n do
        local pos
        if     #i._has  < the.keep      then pos=  1 + (#i._has)
        elseif l.rand() < the.keep/i.n then pos=l.rand(#i._has) end
        if pos then
          i.ok=false -- kept items are no longer sorted
          i._has[pos]=x end end end end end

-- Distance. If missing values, assume max distance.
function Col.dist(i,x,y)
  if x=="?" and y=="?" then return 1 end
  if i.isNom then return x==y and 0 or 1 else
    if     x=="?" then y = Col.norm(i,y); x=y<.5 and 1 or 0
    elseif y=="?" then x = Col.norm(i,x); y=x<.5 and 1 or 0
    else   x,y = Col.norm(i,x), Col.norm(i,y) end
    return math.abs(x-y) end end

-- Diversity: divergence from central tendency (sd,entropy for NOM,RATIO).
function Col.div(i)
  local t = Col.has(i)
  if not i.isNom then return (l.per(t,.9) - l.per(t,.1))/2.58 else
    local e=0
    for _,v in pairs(t) do if v>0 then e=e-v/i.n*math.log(v/i.n,2) end end
    return e end end

-- Sorted contents
function Col.has(i)
  if i.isNom then return i._has else
    if not i.ok then table.sort(i._has) end
    i.ok=true
    return i._has end end

-- Central tendency (mode,median for NOMs,RATIOs)
function Col.mid(i)
  if not i.isNom then return l.per(Col.has(i),.5) else
    local mode,most=nil,-1
    for k,v in pairs(i._has) do if v>most then mode,most=k,v end end
    return mode end end

-- Return num, scaled to 0..1 for lo..hi
function Col.norm(i,x)
  if i.isNom then return x else
    local has= Col.has(i) -- "a" contains all our numbers,  sorted.
    local lo,hi = has[1], has[#has]
    return hi - lo  < 1E-9 and 0 or (x-lo)/(hi-lo) end end

-- Map x to a small range of values. For NOMs, x maps to itself.
function Col.discretize(i,x)
  if i.isNom then return x else
    local has = has(i)
    local lo,hi = has[1], has[#has]
    local b = (hi - lo)/the.bins
    return hi==lo and 1 or math.floor(x/b+.5)*b  end end

--
--      ╔╦╗╔═╗╔╦╗╔═╗
--       ║║╠═╣ ║ ╠═╣
--      ═╩╝╩ ╩ ╩ ╩ ╩
--
-- Holds n records
function Data.new(t) return {rows={}, about=About.new(t) } end

-- Update
function Data.add(i,t) l.push(i.rows, About.add(i.about,t)) end

-- Sort rows, then pretend you didn't
function Data.cheat(i)
  for j,row in pairs(l.sort(i.rows, Row.better)) do
    row.rank = l.rnd(100*j/#i.rows)
    row.evaled= false end
  i.rows = l.shuffle(i.rows) end

-- Replicate structure
function Data.clone(i,  t)
  local out = Data.new(i.about.names)
  for _,row in pairs(t or {}) do Data.add(out,row) end
  return out end

-- Discretize all row values (writing those vals to "cooked").
function Data.discretize(i)
  for _,row in pairs(i.rows) do
    for _,col in pairs(i.about.x) do
      local x = row.cells[col.at]
      if x~= "?" then
        row.cooked[col.at] = Col.discretize(col,x) end end end end

-- Diversity
function Data.div(i) return l.map(i.about.y, Col.div) end

-- Recursively bi-cluster one Data into sub-Datas.
function Data.cluster(i,  rowAbove,stop)
  stop = stop or (#i.rows)^the.Min
  if #i.rows >= 2*stop then
    local A,B,As,Bs,c = Data.half(i.rows,rowAbove)
    i.halves = {c=c, A=A, B=B,
                kids = { Data.cluster(Data.clone(i,As), A, stop),
                         Data.cluster(Data.clone(i,Bs), B, stop) }}end
  return i end

-- Split data according to distance to two  distant points A,B
-- To dodge outliers, don't search all the way to edge (see the.Far).
-- To speed things up:
-- - try to reuse a distant point from above (see rowAbove).
-- - only look at some of the rows (see the.Some).
-- - find distant points in linear time via
--   A=far(any()) and B=far(A).
function Data.half(i, rows,   rowAbove,      c)
  local some= l.many(rows, the.Some)
  local function far(row)
    return l.per(Row.around(row,some), the.Far).row end
  local As,Bs = {},{}
  local A= rowAbove or far(l.any(some))
  local B= far(A)
  local c= Row.dist(A,B)
  local function project(row)
    local a,b = Row.dist(row,A), Row.dist(row,B)
    return {row=row, x=(a^2 + c^2 - b^2)/(2*c)} end
  for n,rowx in pairs(l.sort(l.map(rows, project),l.lt"x")) do
    l.push(n < #rows/2 and As or Bs, rowx.row) end
  return A,B,As,Bs,c end

-- Load from file
function Data.load(sFilename,          data)
  l.csv(sFilename, function(row)
    if data then Data.add(data,row) else data=Data.new(row) end end)
  return data end

-- Central tendency
function Data.mid(i)
  local t={}
  for _,col in pairs(i.about.y) do t.n=#i.rows; t[col.txt] = Col.mid(col) end
  return t end

function Data.best(i,  rowAbove,stop)
  stop = stop or 10 --(#i.rows)^the.Min
  if   #i.rows <= stop
  then return i.rows
  else local A,B,As,Bs,c = Data.half(i, i.rows, rowAbove)
       if   Row.better(A,B)
       then return Data.best(Data.clone(i,As),A,stop)
       else return Data.best(Data.clone(i,l.rev(Bs)),B,stop) end end end

-- Heuristically sort rows by trends in the y-values
-- (specifically, evaluated two remote points, sort worse
-- half by distance to best point, recurse on best half).
function Data.trends(i,out,  rowAbove,stop)
  stop = stop or (#i.rows)^the.Min
  out = out or {}
  if   #i.rows < stop
  then for _,row in pairs(i.rows) do l.push(out,row) end
  else local A,B,As,Bs,c = Data.half(i, i.rows, rowAbove)
       if   Row.better(A,B)
       then for j=#Bs,1,-1 do l.push(out,Bs[j]) end
            Data.trends(Data.clone(i,l.rev(As)), out,A, stop)
       else for _,row in pairs(As) do l.push(out,row) end
            Data.trends(Data.clone(i,Bs), out,B, stop) end end
  return out end

-- ------------------------------------------------------------------------
return RL
```